TinyVM: User's Guide

The information in this guide may only apply to the latest version of TinyVM Unix Edition. For the Cross-Platform Edition, please read the `doc` file included with that distribution.

TinyVM Requirements

These are the current requirements for TinyVM, as far as I can tell. (Please let Jose know if you have been successful with other setups).

- Linux. (See also: Windows, Solaris).
- JDK 1.1.X or JDK 1.2.X. (You can get them at java.sun.com).
- Familiarity with Java or other languages that can run on a JVM.

Installation and Setup

Installation consists of unzipping the file you downloaded. The zip file already contains a top-level directory, so there's no need to create one. Setup consists of the following steps:

- Set **RCXTTY** to the IR serial port (e.g. /dev/ttyS1).
- Go to the installation directory and run `make`.
- Add the `bin` directory to your **PATH** setting. Remove previous versions of TinyVM from your **PATH**.

Additionally, the JDK's bin directory is assumed to be part of your **PATH** setting.

Downloading the Firmware

To download the firmware, run:

```
tvmfirmdl
```

I have also tried Dave Baum's firmware downloader included with NQC . You should be able to use it for fast downloads, as follows:

```
nqc -firmfast $TINYVM_HOME/bin/tinyvm.srec
```

Donwloading the firmware will take a couple of minutes. The counter will reach 1020 or so. Then you'll hear 2 beeps, and the LCD will show a number in the 200s or 300s. This means TinyVM's firmware is in program-download mode.

The number shown in the LCD is the battery power level. For fresh batteries, it should be around 333. I've had TinyVM work with a battery power level of 212, but I do not recommend using it if the power level is below 250.

TinyVM allows other firmware to get downloaded when it is in program-download mode. (You can always get to program-download mode by switching the RCX off and on).

RCX Java Programs

Writing Java programs for TinyVM is not unlike writing programs for other Java environments. However, you should keep in mind that only a small subset of the standard Java APIs are supported by TinyVM. When in doubt, consult TinyVM's API Documentation.

Compiling

To compile Java programs for TinyVM you should use `tvmc` instead of `javac`. Note that `tvmc` is simply a `javac` invocation script. It's essentially equivalent to:

```
javac -classpath $CLASSPATH:$TINYVM_HOME/lib/classes.jar <class>.java
```

NOTE: Do *not* add classes.jar to your CLASSPATH setting. This will affect other JVMs.

If you try to compile a TinyVM program using only `javac`, the compiler will not be able to find TinyVM APIs, which are located under lib/classes.jar. If you try to place classes.jar in the CLASSPATH before calling javac, some of the core Java classes from classes.jar might be confused with core Java classes from the JDK's classes.zip file.

Some classes from the JDK's classes.zip (or rt.jar) file might work with TinyVM, as long as no native methods are involved. However, this hasn't been tested and it is not recommended.

Linking

The linking step is one that departs from what you normally would do in other Java environments. The linker in TinyVM produces a binary file containing all the program code that needs to be downloaded into the RCX. Dynamic loading of classes was not considered a good idea in the design of TinyVM, because it would not allow robots to be independent of the development PC. In order to link your program, use

```
tvmld -o <class>.tvm <class>
```

This will produce a `.tvm` file which you can download into the RCX.

Downloading Java Programs

TinyVM's firmware must be in program-download mode (two beeps, and battery power level shown) before you can download any programs into it. If you have another program running in TinyVM, switch off the RCX and then switch it on.

To download a previously linked program, run:

```
tvm <class>.tvm
```

The linker (`tvmld`) can download programs directly to the RCX. Simply pass a `-d` option to it, instead of `-o`, as follows:

```
tvmld <class> -d
```

You should see the number of bytes downloaded divided by 100 in the LCD. If nothing happens, make sure the RCX is in range, and try again. If you hear beeps, kill tvm, switch off the RCX, wait until the IR tower's light is off, and try again.

Exceptions

TinyVM will show simple exception/error "traces" for all uncaught exceptions/errors. They will be displayed in the LCD using two numbers: (1) signature of the method where the throwable object was thrown, and (2) index of the object's class `mod` 10. A list of classes and signatures are dumped by the linker (tvmld) if you pass a -verbose option to it.

To take an example,

[0000 7]

could mean that a NullPointerException exception (class index 7) was thrown in a main method (method signature 0). To facilitate identification of stack traces, you will also hear a low buzz (like system sound 4) and the LCD will show a wedge right below sensor 2.

NOTE: Uncaught exception traces will suspend your program. To continue, press On/Off.

Performance Tips

- Declare your methods to be either `private`, `final` or `static` whenever possible.
- The location of non-private virtual methods matters: place your virtual methods close to the top of the class declaration. The same goes for static initializers and `main` methods.
- Local variable access is considerably faster than field access. This even applies to constants.
- Accessing array elements of arrays assigned to local variables is faster than field access, but slightly slower than local variable access.

Questions

You can find additional information in the README file. Feel free to contact TinyVM's author, Jose Solorzano, at jhsolorz@yahoo.com, if you have any questions not answered in the documentation, or if you'd like to contribute to TinyVM in any way.

TinyVM: Technical Notes

The following information has been updated to apply to TinyVM 0.1.1. It may or may not apply to the latest downloadable version of TinyVM.

Footprint

TinyVM's firmware occupies about 9800 bytes of RAM in the RCX. As points of reference, consider Lego's official firmware, which occupies about 16Kb of RAM, and legOS, which requires about 8Kb.

The amount of usable memory available to the firmware is close to 28 Kb. So there is about 18 Kb of memory TinyVM can use to store programs and data. This is confirmed by a test program in TinyVM's distribution (regression/Test13.java) which is able to allocate 4338 objects (17352 bytes) before throwing an `OutOfMemoryError` error.

Programs can be as small as half a Kb, but they will usually be close to 2 Kb if they use essential RCX APIs. Program size does not grow dramatically as you write more code.

To reduce program download time, classes that are always required will be predownloaded to the RCX in a future version of TinyVM. (At least that's the plan).

Linker

Classes are *not* objects in TinyVM. If you have tried out TinyVM, you are probably already familiar with its linker. The function of the linker is to resolve all references in advance, and to create a binary image that is ready to be copied into TinyVM's internal data structures.

This part of the design helps make TinyVM small (and actually feasible), because it doesn't require any class file parsing/loading code to be running in the RCX. Additionally, the format of the linked binary file is such that it doesn't require any class or component names to be kept, so it results in very small downloads (if you compare them to the class files they came from).

Clearly, a disadvantage of this design is that it doesn't allow for the implementation of dynamic class loading or reflection.

Memory Management

Memory allocation in TinyVM is as simple as it can be: A list of free blocks is maintained by the system. Each free block has a 4-byte header: two bytes for the block size, and two bytes for the next-block offset. (An offset was used instead of an address to allow TinyVM to run on 32-bit systems). All allocated blocks start at an even address.

Deallocation routines exist in TinyVM, but they are only used to deallocate thread stacks. TinyVM does not have garbage collection as of this writing. Given the fact that a small program can create more than 4000 objects before it runs out of memory, it seems that most RCX programs could be written without the need for garbage collection. Nevertheless, users have to be cautions about creating a bounded number of objects, i.e. creating objects in initializers or sections of code that will only execute a bounded number of times. Object recycling is a technique that could come in handy. An advantage of not having garbage collection is that timing is more deterministic.

Implementing garbage collection in TinyVM would not be difficult, but it would probably add about 1 Kb of code. The main tasks involved would be: (1) writing the memory traversal code, and (2) marking reference entries in all stacks.

Objects

All objects in TinyVM have a four-byte header, which includes: class index (1 byte), thread index (1 byte), sychronization count (1 byte), and additional bits needed to check if that part of memory is actually a free block or an array. In the case of arrays, the class index is replaced by the length of the array and its element type (1 byte and 4 bits, respectively). As such, the maximum array length allowed per dimension is 255. Arrays are assumed to be instances of `java.lang.Object`.

Fields in objects are packed, e.g. a `byte` field occupies exactly one byte; an `int` field occupies four bytes, and it is *not* aligned. This may hurt performance somewhat, but it saves space. A general theme of TinyVM's design is to place footprint considerations before performance.

Multi-Threading

TinyVM implements a very simple preemptive multi-threading scheme. Each thread object has a pointer to the next thread in a global list of threads. The list is circular, and the scheduler consists of a single function: `switch_thread()`.

Time-slicing is actually *not* the technique of choice. Scheduling is done based on number of opcodes executed. The reason for this design choice is mainly simplicity, but also to make sure that certain operations are atomic as far as programs are concerned.

Currently, the number of opcodes that a thread is allowed to execute before it's preempted is 128. This is based on an educated guess of what may be acceptable, but seems to work fairly well. The value can be configured by modifying a constant in vmsrc/configure.h and recompiling the VM.

Each thread has an ID, used by monitors (objects) to refer to their owning threads. If the number of created threads reaches 255, the system will throw an `OutOfMemoryError` error. This would happen upon calling `Thread.start()`.

Synchronization

Synchronization is supported, except for `static synchronized` methods. As stated, classes are not objects, so there's nothing to synchronize such methods with. (Incidentally, certain JDK VMs used to have a bug that could deadlock a program with `static synchronized` methods, and there are code-checking tools that warn about their use). The alternative is to use `synchronized` blocks.

Each monitor (object) has a byte whose value is either 0, indicating that it isn't owned, or the thread ID of its owner. A monitor also contains a one-byte synchronization count, which is used to count the number of times a synchronized block has been reentered in a single thread of execution. In theory, a program will start to misbehave if a synchronized block has been reentered more than 255 times, but this is a minor issue in TinyVM since the stack is defined by default to allow considerably less levels of recursion.

Interpreter Architecture

TinyVM's interpreter was designed to make it impossible (or very difficult) to overflow the native stack. In other words, recursion was avoided in TinyVM's code at all costs. (This is not true of multi-dimensional array allocation, but the problem can be resolved by making the linker reject programs with arrays of more than N dimensions). Essentially, there's a method called `engine` in `vmsrc/interpreter.c`, which loops on a switch statement. No part of `engine` causes `engine` to be called recursively.

Method Dispatch

Dispatch of static and special methods consists of allocating a new stack frame in the stack frame array, and setting the interpreter's PC and stack pointers as required. The top words of the caller's operand stack (i.e. method arguments) are *not* copied to the callee's local variable stack. Instead, the callee's local variable stack is set to point to the first parameter in the caller's operand stack. In other words, the caller and the callee share part of their stacks.

Dispatch of virtual methods was written to be as simple as possible (again, sacrificing performance for footprint). Essentially, an object's class is scanned sequentially until a method matching the call signature is found. If one isn't found, its superclass is examined, and so on.

Users are encouraged to declare methods as `final`, `private` or `static` whenever possible, because these methods are not dispatched using the virtual mechanism just described.

Native Methods

JNI was not even seriously considered in the design of TinyVM. It would require considerable amounts of extra code and data structures that are not compatible with TinyVM's design goals.

A very simple mechanism is used to implement native calls: The linker and the VM agree in advance about a set of native method signatures (see common/signatures.db). These signatures are actually hard-wired into each of these programs. In the VM, the signatures are translated into constant names, in a manner that is not completely robust (see vmsrc/specialsignatures.h). There could be name collisions if two native methods happen to have the same name, ignoring case, to take an example. The VM dispatches each native call using a switch statement on the method signature (see vmsrc/native.c).

As such, it's trivial to add a native method to TinyVM: (1) The native method's signature is defined in common/signatures.db; (2) A corresponding `case` is added to vmsrc/native.c and vmtest/nativeemul.c; and (3) The method is declared as `native` in any Java class.

Since the mapping of native methods is based on method signatures, two classes can actually contain the same native method. This may come in handy sometimes, but it's probably unsafe from the user's point of view.

Comments

Evidently, there are many areas where TinyVM could improve. Please send any ideas or comments you may have to TinyVM's author, Jose Solorzano, at jhsolorz@yahoo.com.